Remarks

This reply is in response to the Office communication dated April 1, 2005. Unless otherwise indicated, page references are to that communication.

Claim 1 has been amended to recite that the single hardware instruction referenced there is capable of processing a plurality of characters "in a single invocation of the instruction". This accurately characterizes the Translate Two to One (TRTO) instruction referenced in the specification, as well as in publicly available documentation such as the z/Architecture Principles of Operation, SA22-7832-00 (Dec. 2000), an excerpt of which (pages 3 to 4 and 7-152 to 7-157) accompanies this amendment as an attachment. As described on page 7-153 of that publication, on execution of a TRTO instruction specifying first and second operands:

> The characters of the second operand are used as arguments to select function characters from a translation table designated by the address in general register 1. Each function character selected from the translation table is compared to a test character in general register 0, and, unless an equal comparison occurs, is placed at the first-operand location. The operation proceeds until a selected function character equal to the test character is encountered, the end of the second operand is reached, or a CPU-determined number of characters have been processed, whichever occurs first.

Claim 14 has been amended to recite that the sub-codepages "other than said highest-priority sub-codepage" comprise a first set of one or more higher-priority sub-codepages and a second set of one or more lower-priority sub-codepages. While this limitation is believed to have been implicit in the claim as previously presented, it is now made explicit.

Claims 1-13

Claims 1-13 stand rejected under 35 U.S.C. §§ 102(e) and 103(a) as being either anticipated by or unpatentable over U.S. Patent 6,204,782 to Gonzalez et al. ("Gonzalez").

Claim 1 as currently amended is directed to a method for converting a source string in which the characters are converted using a single hardware instruction capable of processing a plurality of characters <u>in a single invocation of the instruction</u> (emphasis added).

The Examiner contends, in his response to applicant's previous arguments, that Gonzalez "in fact does disclose the claimed limitation of using a single hardware instruction capable of processing a plurality of characters" (page 8, ¶ 8). As evidence of this, the Examiner points to the flow diagram of Fig. 7, as well as the "hardware implementation" shown in Fig. 10. The Examiner concludes from Fig. 7 that "[i]t is clear . . . that only a single hardware instruction is needed to start the process of converting the source string into the target string because the text processing automatically continues until the entire string is converted" (pages 8-9). Applicant respectfully disagrees.

The Examiner will note that his conclusion—that only a single hardware instruction is needed <u>to start the process</u> of converting the source string into the target string—does not quite match the assertion made elsewhere—namely, that Gonzalez discloses a single hardware instruction capable of <u>processing</u> a plurality of characters. The mere fact that a single hardware instruction is capable of <u>starting</u> a conversion process says nothing about whether that same instruction is capable of <u>completing</u> the conversion process. In fact, it would appear that Gonzalez requires a multiplicity of instruction invocations to complete even <u>a single loop</u> of the procedure shown in Fig. 7. He would require even more instruction invocations, of course, to perform the entire conversion process comprising multiple traversals of the loop shown in Fig. 7.

The Examiner refers to the "hardware implementation" shown in Fig. 10. However, that implementation is merely a general-purpose computer, with no special enhancements for performing the disclosed method. And while the patentee notes that the computer system "may be specially constructed for the required purposes" (col. 19, lines 56-57), he gives no examples of any such specially constructed system, much less an example using a particular machine instruction.

For the foregoing reasons, applicant respectfully submits that claim 1 as amended and the claims dependent thereon distinguish patentably over the art cited by the Examiner.

Claims 14-20

Claims 14-20 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Gonzalez.

Claims 14-20 as amended are directed to that feature of applicant's invention whereby the sub-codepages other than the highest-priority sub-codepage comprise a first set of one or more higher-priority sub-codepages and a second set of one or more lower-priority sub-codepages. If a character is found in a sub-codepage belonging to the first set of sub-codepages (i.e., the higher-priority sub-codepages), the conversion process is continued with that sub-codepage as the current sub-codepage. On the other hand, if the character is found in a sub-codepage belonging to the second set of sub-codepages, the character is converted using that sub-codepage and the conversion process is continued with the highest-priority sub-codepage as the current sub-codepage.

In rejecting these claims, the Examiner seems to argue that claim 14 as previously presented reads on Gonzalez's first approach of always reverting to the default codepage after a successful conversion, since the default codepage could be regarded as a constituting the only member of a first set of code pages.[1] It would have been obvious, the Examiner argues, "to have modified Gonzalez . . . to have created a high-priority sub-code page set instead of only a highest-priority sub-codepage so that a block of source text including two major sub-codepages could have translated both of the high-priority sub-codepages of the source text string according to the first approach as taught by Gonzalez in fig. 7 and col. 15 line 32 - col. 17 line 45" (page 6). Applicant respectfully disagrees.

[1] It is unclear to applicant whether the Examiner is saying that Gonzalez anticipates claim 14 (because the highest-priority sub-codepage is a set of one) or merely renders it obvious (because one could enlarge the set to more than one). Applicant has never argued, however, that a "set" requires more than one member, and the claims as currently worded (". . . set of <u>one</u> or more . . .") imply that one member suffices. In any event, applicant has amended claim 14 to specify that the sub-codepages other than the first sub-codepage comprise the first and second sets.

In describing his approaches, the patentee distinguishes between a bias toward "a preferred target encoding" (emphasis added) and attempting to minimize switching between target encodings (col. 6, lines 45-51). There is no notion here of having more than a single "preferred" target encoding for the purposes of minimizing switching between encodings. Rather, the presumption is that the first encoding differs in kind from all lower-priority encodings, since this is where one always goes after an unsuccessful conversion attempt. Indeed, the only suggestion in the record for having more than a single "preferred" encoding comes from applicant's own disclosure and not Gonzalez or any other cited art. This is clearly hindsight, however, and thus impermissible.

For the foregoing reasons, applicant respectfully submits that claims 14-20 distinguish patentably over the art cited by the Examiner.

Conclusion

Reconsideration of the application as amended is respectfully requested. It is hoped that upon such consideration the Examiner will hold all claims allowable and pass the case to issue at an early date. Such action is earnestly solicited.

Respectfully submitted,
JOACHIM M. BAUER

By

William A. Kinnaman, Jr.
Registration No. 27,650
Phone: (845) 433-1175
Fax: (845) 432-9601

WAK/wak

z/Architecture

IBM

Principles of Operation

SA22-7832-00

Note:

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xv.

Softcopy Note:

The reader should be aware of the fact that this publication contains many symbols, such as superscripts, that may not display correctly with any given hardware or software. The definitive version of this publication is the hardcopy version.

First Edition (December 2000)

This publication is provided for use in conjunction with other relevant IBM publications, and IBM makes no warranty, express or implied, about its completeness or accuracy. The information in this publication is current as of its publication date but is subject to change without notice.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to:

International Business Machines Corporation
Department 55JA Mail Station P384
2455 South Road
Poughkeepsie, N.Y., 12601-5400
United States of America

FAX (United States & Canada): 845+432-9405
FAX (Other Countries): Your International Access Code+1+845+432-9405
IBMLink (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.ibm.com/s390/os390/webqs.html

31-bit mode, may be set to zeros or may remain unchanged from their original values.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1 and $R_1 + 1$ always remain unchanged.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R_2 register is the same register as the R_1 or $R_1 + 1$ register, the results are unpredictable.

When R_1 or R_2 is zero, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portion of the first operand to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

Access exceptions for all 256 bytes of the second operand may be recognized, even if not all bytes are used.

Access exceptions are not recognized if the R_1 field is odd. When the length of the first operand is zero, no access exceptions for the first operand are recognized.

Resulting Condition Code:

0 Entire first operand processed without finding a byte equal to the test byte
1 First-operand byte is equal to the test byte
2 --
3 CPU-determined number of bytes processed

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Specification

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the translation. The program need not determine the number of bytes that were translated.
2. The instruction can improve performance by being used in place of a TRANSLATE AND TEST instruction that locates an escape character, followed by a TRANSLATE instruction that translates the bytes preceding the escape character.
3. The storage operand references of TRANSLATE EXTENDED may be multiple-access references. (See "Storage-Operand Consistency" on page 5-86.)

TRANSLATE ONE TO ONE

TROO R_1,R_2 [RRE]

'B993'	////////	R_1	R_2

0 16 24 28 31

TRANSLATE ONE TO TWO

TROT R_1,R_2 [RRE]

'B992'	////////	R_1	R_2

0 16 24 28 31

TRANSLATE TWO TO ONE

TRTO R_1,R_2 [RRE]

'B991'	////////	R_1	R_2

0 16 24 28 31

TRANSLATE TWO TO TWO

TRTT R_1,R_2 [RRE]

'B990'	////////	R_1	R_2
0	16	24	28 31

The characters of the second operand are used as arguments to select function characters from a translation table designated by the address in general register 1. Each function character selected from the translation table is compared to a test character in general register 0, and, unless an equal comparison occurs, is placed at the first-operand location. The operation proceeds until a selected function character equal to the test character is encountered, the end of the second operand is reached, or a CPU-determined number of characters have been processed, whichever occurs first. The result is indicated in the condition code.

The lengths of the operand and test characters are as follows:

- For TRANSLATE ONE TO ONE, the second-operand, first-operand, and test characters are single bytes.
- For TRANSLATE ONE TO TWO, the second-operand characters are single bytes, and the first-operand and test characters are double bytes.
- For TRANSLATE TWO TO ONE, the second-operand characters are double bytes, and the first-operand and test characters are single bytes.
- For TRANSLATE TWO TO TWO, the second-operand, first-operand, and test characters are double bytes.

For TRANSLATE ONE TO ONE and TRANSLATE TWO TO ONE, the test character is in bit positions 56-63 of general register 0. For TRANSLATE ONE TO TWO and TRANSLATE TWO TO TWO, the test character is in bit positions 48-63 of general register 0.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the second-operand location is specified by the contents of bit positions 32-63 of general register $R_1 + 1$, and those contents are treated as a 32-bit unsigned binary integer. In the 64-bit addressing mode, the number of bytes in the second-operand location is specified by the contents of bit positions 0-63 of general register $R_1 + 1$, and those contents are treated as a 64-bit unsigned binary integer. The length of the first-operand location is considered to be the same as that of the second operand for TRANSLATE ONE TO ONE and TRANSLATE TWO TO TWO, twice that for TRANSLATE ONE TO TWO, and one half that for TRANSLATE TWO TO ONE.

For TRANSLATE TWO TO ONE and TRANSLATE TWO TO TWO, the length in general register $R_1 + 1$ must be an even number of bytes; otherwise, a specification exception is recognized.

The translation table is treated as being on a doubleword boundary for TRANSLATE ONE TO ONE and TRANSLATE ONE TO TWO and on a 4K-byte boundary for TRANSLATE TWO TO ONE and TRANSLATE TWO TO TWO. The rightmost bits of the register that are not used to form the address, which are bits 61-63 in the doubleword case and bits 52-63 in the 4K-byte case, are ignored.

The handling of the addresses in general registers R_1, R_2, and 1 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 and 40-60 or 40-51 of 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of registers R_1 and R_2 and 33-60 or 33-51 of 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of registers R_1 and R_2 and 0-60 or 0-51 of 1 constitute the address.

The contents of the registers just described are shown in Figure 7-24 on page 7-154.

In the access-register mode, the contents of access registers R_1, R_2, and 1 are used for accessing the first operand, second operand, and translation table, respectively.

The length of the translation table designated by the address contained in general register 1 is as follows:

- For TRANSLATE ONE TO ONE, the translation-table length is 256 bytes; each of the 256 function characters is a single byte.
- For TRANSLATE ONE TO TWO, the translation-table length is 512 bytes; each of the 256 function characters is a double byte.
- For TRANSLATE TWO TO ONE, the translation-table length is 65,536 (64K) bytes; each of the 64K function characters is a single byte.
- For TRANSLATE TWO TO TWO, the translation-table length is 131,072 (128K) bytes; each of the 64K function characters is a double byte.

The characters of the second operand are selected one by one for translation, proceeding left to right. Each argument character is added to the initial translation-table address. The addition is performed following the rules for address arith-

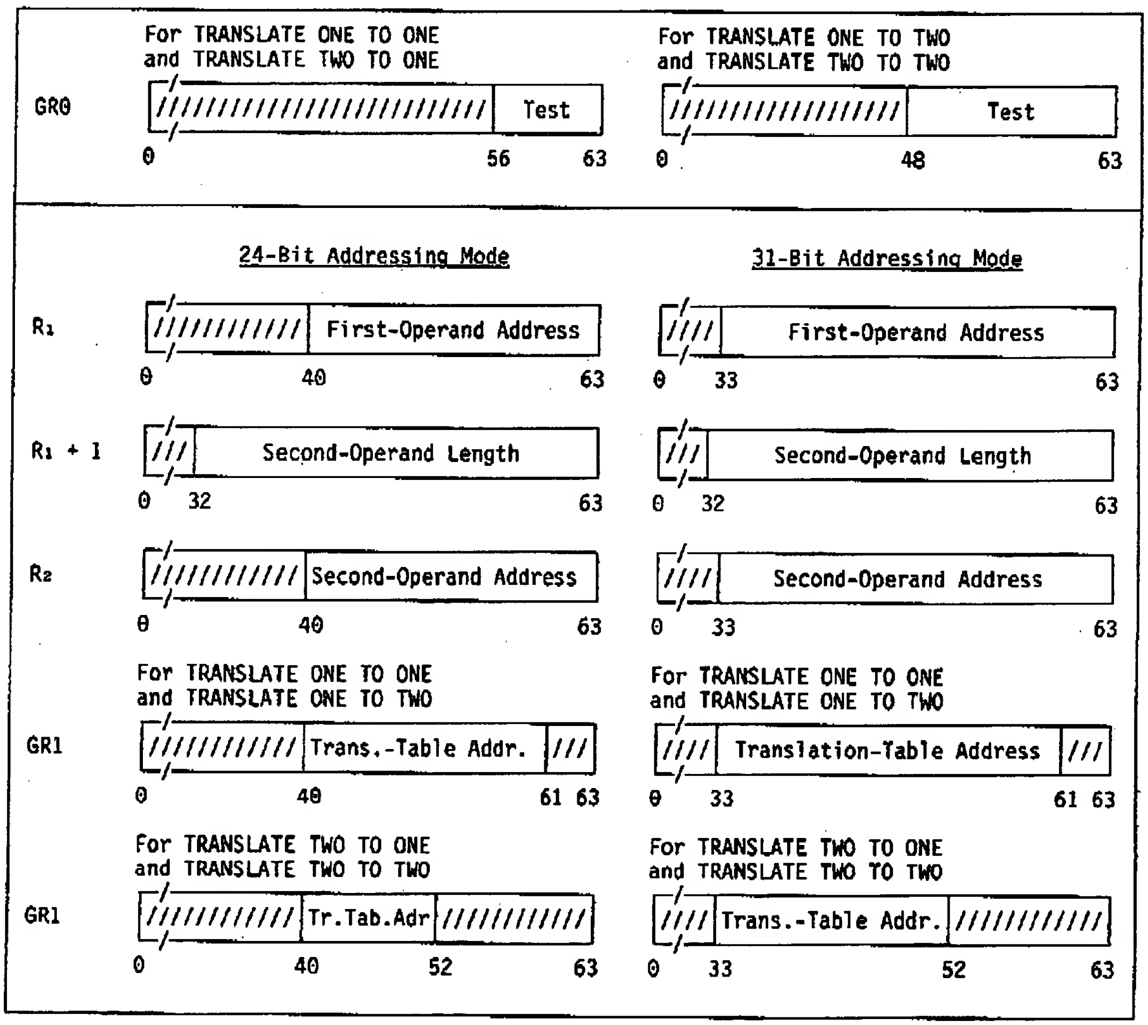

Figure 7-24 (Part 1 of 2). Register Contents for TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO

metic, with the argument character treated as follows:

- For TRANSLATE ONE TO ONE, the argument character is treated as an eight-bit unsigned binary integer extended on the left with 56 zeros.
- For TRANSLATE ONE TO TWO, the argument character is treated as an eight-bit unsigned binary integer extended on the right with a zero and on the left with 55 zeros.
- For TRANSLATE TWO TO ONE, the argument character is treated as a 16-bit unsigned binary integer extended on the left with 48 zeros.
- For TRANSLATE TWO TO TWO, the argument character is treated as a 16-bit unsigned binary integer extended on the right with a zero and on the left with 47 zeros.

The rightmost bits of the translation-table address that are ignored (61-63 or 52-63) are treated as zeros during this addition.

The sum is used as the address of the function character.

Each function character selected as described above is first compared to the test character in general register 0. If the result is an equal comparison, the operation is completed. If the function character is not equal to the test character, the function character is placed in the next available character position in the first operand, that is, the first function character is placed at the beginning of the first-operand location, and each successive function character is placed immediately to the right of the preceding character. The second operand and the translation table are not altered unless an overlap occurs.

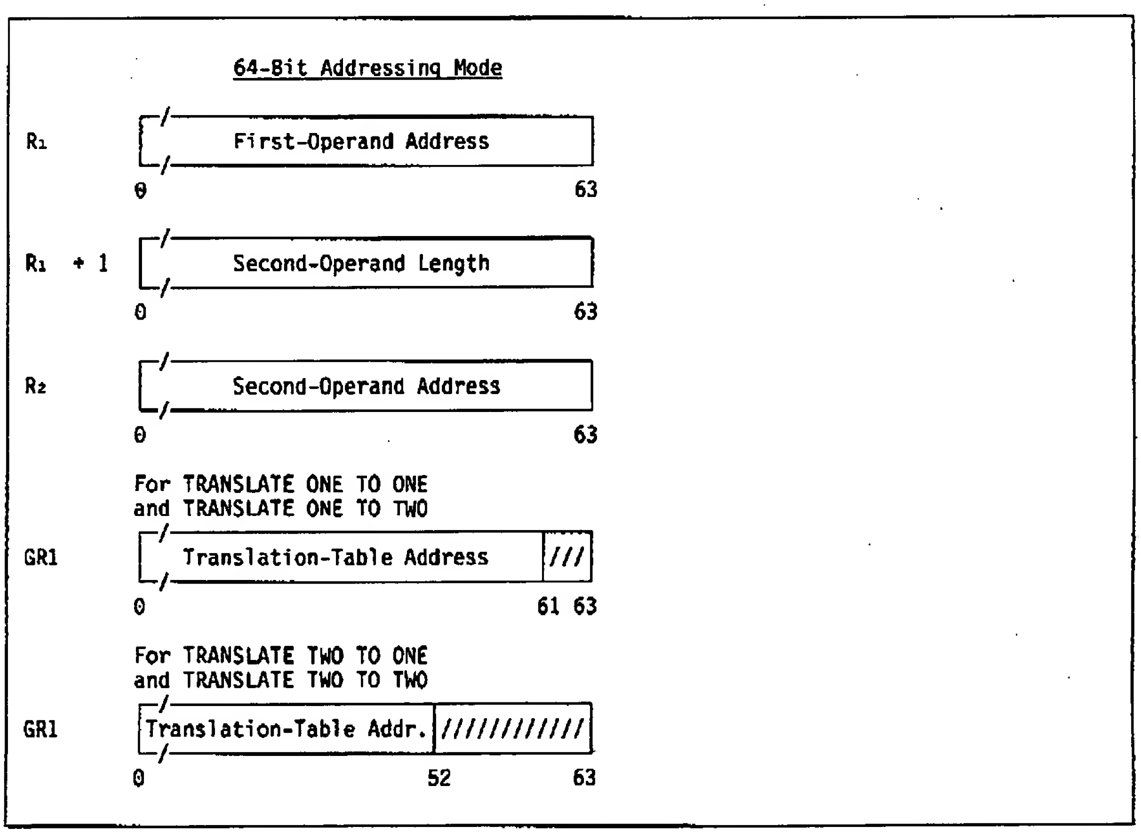

Figure 7-24 (Part 2 of 2). Register Contents for TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO

The operation proceeds until a selected function character equal to the test character is encountered, the second-operand location is exhausted, or a CPU-determined number of second-operand characters have been processed.

When a selected function character equal to the test character is encountered, condition code 1 is set. When the second-operand location is exhausted without finding a selected function character equal to the test character, condition code 0 is set. When a CPU-determined number of characters have been processed, condition code 3 is set. Condition code 3 may be set even when the next character to be processed results in a function character equal to the test character or when the second-operand location is exhausted. In these cases, condition code 1 or 0, respectively, will be set when the instruction is executed again.

If the operation is completed with condition code 0, the contents of general register R_2 are incremented by the contents of general register $R_1 + 1$, and the contents of general register R_1 are incremented as follows:

- For TRANSLATE ONE TO ONE and TRANSLATE TWO TO TWO, the same as for general register R_2.
- For TRANSLATE ONE TO TWO, by twice the amount for general register R_2.
- For TRANSLATE TWO TO ONE, by one half the amount for general register R_2.

The contents of general register $R_1 + 1$ are then set to zero.

If the operation is completed with condition code 1, the contents of general register $R_1 + 1$ are decremented by the number of second-operand bytes processed before the character that selected a function character equal to the test character was encountered, and the contents of general register R_2 are incremented by the same number, so that general register R_2 contains the address of the character that selected a function character equal to the test character. The contents of general register R_1 are incremented by the same, twice, or one half the number, as described above for condition code 0.

If the operation is completed with condition code 3, the contents of general register $R_1 + 1$ are decremented by the number of second-operand bytes processed, and the contents of general register R_2 are incremented by the same number, so that the instruction, when reexecuted, contains the address of the next character to be processed. The contents of general register R_1 are incremented by the same, twice, or one half the number, as described above for condition code 0.

When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, the bits in bit positions 32-39 of them that are not part of the address may be set to zeros or may remain unchanged from their original values. In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, and R_2 always remain unchanged.

The contents of general registers 0 and 1 remain unchanged.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

During instruction execution, CPU retry may result in condition code 3 being set with possibly incorrect data having been stored in the first operand location at or to the right of the location designated by the final address in general register R_1. The amount of data stored depends on the operation and the point in time at which CPU retry occurred. In all cases, the storing will occur again, with correct data stored, when the instruction is executed again to continue processing the same operands.

When the R_1 register is the same register as the R_2 register, the R_1 or R_2 register is register 0, or the R_2 register is register 1, the results are unpredictable.

When any of the first and second operands and the translation table overlaps another of them, the results are unpredictable.

Access exceptions for the portion of the first or second operand to the right of the last character processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last character processed.

Access exceptions for all characters of the translation table may be recognized even if not all characters are used.

Access exceptions are not recognized if the R_1 field is odd. When the length of the second operand is zero, no access exceptions for the first or second operand are recognized, and access exceptions for the translation table may or may not be recognized.

Resulting Condition Code:

0 Entire second operand processed without finding a resulting function character equal to the test character
1 Second-operand character found resulting in a function character equal to the test character
2 --
3 CPU-determined number of characters processed

Program Exceptions:

- Access (fetch, operand 2 and translation table; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

Programming Notes:

1. These instructions differ from the TRANSLATE EXTENDED instruction by having the following attributes:
   - Depending on the instruction used, the sets of argument characters and function characters each can contain single-byte or double-byte characters.
   - The test character is compared to a resulting function character instead of to an argument character.
   - The argument (source) and function (destination) operands are different operands.
2. When condition code 3 is set, the program can simply branch back to the instruction to continue the translation. The program need not determine the number of characters that were translated.
3. The storage operand references of these instructions may be multiple-access references. (See "Storage-Operand Consistency" on page 5-86.)

UNPACK

UNPK $D_1(L_1,B_1),D_2(L_2,B_2)$ [SS]

'F3'	L_1	L_2	B_1	D_1	B_2	D_2
0	8	12	16	20	32	36 47

The format of the second operand is changed from packed to zoned, and the result is placed at the first-operand location. The packed and zoned formats are described in Chapter 8, "Decimal Instructions."

The second operand is treated as having the packed format. Its digits and sign are placed unchanged in the first-operand location, using the zoned format. Zone bits with coding of 1111 are supplied for all bytes except the rightmost byte, the zone of which receives the sign of the second operand. The sign and digits are not checked for valid codes.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first-operand field is too short to contain all digits of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if the operands were processed one byte at a time and as if the first result byte were stored immediately after fetching the first operand byte. The entire rightmost second-operand byte is used in forming the first result byte. For the remainder of the field, information for two result bytes is obtained from a single second-operand byte, and execution proceeds as if the leftmost four bits of the byte were to remain available for the next result byte and need not be refetched. Thus, the result is as if two result bytes were to be stored immediately after fetching a single operand byte.

Condition Code: The code remains unchanged.